UNITED STATES PATENT APPLICATION

DECOUPLED STORE ADDRESS AND DATA IN A MULTIPROCESSOR SYSTEM

INVENTORS

STEVEN L. SCOTT
of Eau Claire, Wisconsin

GREGORY J. FAANES
of Eau Claire, Wisconsin

Schwegman, Lundberg, Woessner, & Kluth, P.A.
1600 TCF Tower
121 South Eighth Street
Minneapolis, Minnesota 55402
ATTORNEY DOCKET 1376.697US1

DECOUPLED STORE ADDRESS AND DATA IN A MULTIPROCESSOR SYSTEM

Related Applications

This application is related to U.S. Patent Application No. _____, entitled "Multistream Processing System and Method", filed on even date herewith; to U.S. Patent Application No. _____, entitled "System and Method for Synchronizing Memory Transfers", Serial No. _____, filed on even date herewith; to U.S. Patent Application No. _____, entitled "Decoupled Vector Architecture", filed on even date herewith; to U.S. Patent Application No. _____, entitled "Latency Tolerant Distributed Shared Memory Multiprocessor Computer", filed on even date herewith; to U.S. Patent Application No. _____, entitled "Relaxed Memory Consistency Model", filed on even date herewith; to U.S. Patent Application No. _____, entitled "Remote Translation Mechanism for a Multinode System", filed on even date herewith; and to U.S. Patent Application No. _____, entitled "Method and Apparatus for Local Synchronizations in a Vector Processor System", filed on even date herewith, each of which is incorporated herein by reference.

Field of the Invention

The present invention is related to multiprocessor computers, and more particularly to a system and method for decoupling a write address from write data.

Background Information

As processors run at faster speeds, memory latency on accesses to memory looms as a large problem. Commercially available microprocessors have addressed this problem by decoupling memory access from manipulation of the data used in that memory reference. For instance, it is common to decouple memory references from execution based on those references and to decouple address computation of a memory reference from the memory reference itself. In addition, Scalar processors already

decouple their write addresses and data internally. Write addresses are held in a "write buffer" until the data is ready, and in the mean time, read requests are checked against the saved write addresses to ensure ordering.

With the increasing pervasiveness of multiprocessor systems, it would be beneficial to extend the decoupling of write addresses and write data across more than one processor, or across more than one functional unit within a processor. What is needed is a system and method of synchronizing separate write requests and write data across multiple processors or multiple functional units within a microprocessor which maintains memory ordering without collapsing the decoupling of the write address and the write data.

Brief Description of the Drawings

Fig. 1a illustrates a multiprocessor computer system according to the present invention;

Fig. 1b illustrates another example of a multiprocessor computer system according to the present invention;

Fig. 2 illustrates a method of decoupling store address and data in a multiprocessor system according to the present invention;

Fig. 3 illustrates a processor having a plurality of processing units connected to a shared memory according to the present invention; and

Fig. 4 illustrates a processor node having a plurality of processors connected to a shared memory according to the present invention.

Description of the Preferred Embodiments

In the following detailed description of the preferred embodiments, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention.

A multiprocessor computer system 10 is shown in Fig. 1a. Multiprocessor computer system 10 includes N processors 12 (where N>1) connected by a interconnect network 18 to a shared memory 16. Shared memory 16 includes a store address buffer 19.

Not all processors 16 have to be the same. A multiprocessor computer system 10 having different types of processors connected to a shared memory 16 is shown in Fig. 1b. Multiprocessor computer system 10 includes a scalar processing unit 12, a vector processing unit 14 and a shared memory 16. Shared memory 16 includes a store address buffer 19.

In the example shown, scalar processing unit 12 and vector processing unit 14 are connected to memory 16 across an interconnect network 18. In one embodiment, vector processing unit 14 includes a vector execution unit 20 connected to a vector load/store unit 22. Vector load/store unit 22 handles memory transfers between vector processing unit 14 and memory 16.

The vector and scalar units in vector processing computer 10 are decoupled, meaning that scalar unit 12 can run ahead of vector unit 14, resolving control flow and doing address arithmetic. In addition, in one embodiment, computer 10 includes load buffers. Load buffers allow hardware renaming of load register targets, so that multiple loads to the same architectural register may be in flight simultaneously. By pairing vector/scalar unit decoupling with load buffers, the hardware can dynamically unroll loops and get loads started for multiple iterations. This can be done without using extra architectural registers or instruction cache space (as is done with software unrolling and/or software pipelining). These methods of decoupling are discussed in Patent Application No. xx/yyyyy, entitled "Decoupled Vector Architecture", filed on even date herewith, the description of which is incorporated herein by reference.

In one embodiment, both scalar processing unit 12 and vector processing unit 14 employ memory/execution decoupling. Scalar and vector loads are issued as soon as possible after dispatch. Instructions that depend upon load values are dispatched to

queues, where they await the arrival of the load data. Store addresses are computed early (in program order interleaving with the loads), and their addresses saved for later use.

Methods of memory/execution decoupling are discussed as well in Patent Application No. xx/yyyyy, entitled "Decoupled Vector Architecture", filed on even date herewith, the description of which is incorporated herein by reference.

In one embodiment, each scalar processing unit 12 is capable of decoding and dispatching one vector instruction (and accompanying scalar operand) per cycle. Instructions are sent in order to the vector processing units 14, and any necessary scalar operands are sent later after the vector instructions have flowed through the scalar unit's integer or floating point pipeline and read the specified registers. Vector instructions are not sent speculatively; that is, the flow control and any previous trap conditions are resolved before sending the instructions to vector processing unit 14.

The vector processing unit renames loads only (into the load buffers). Vector operations are queued, awaiting operand availability, and issue in order. No vector operation is issued until all previous vector memory operations are known to have completed without trapping (and as stated above, vector instructions are not even dispatched to the vector unit until all previous scalar instructions are past the trap point). Therefore, vector operations can modify architectural state when they execute; they never have to be rolled back, as do the scalar instructions.

In one embodiment, scalar processing unit 12 is designed to allow it to communicate with vector load/store unit 22 and vector execution unit 20 asynchronously. This is accomplished by having scalar operand and vector instruction queues between the scalar and vector units. Scalar and vector instructions are dispatched to certain instruction queues depending on the instruction type. Pure scalar instructions are just dispatched to the scalar queues where they are executed out of order. Vector instructions that require scalar operands are dispatched to both vector and scalar instruction queues. These instructions are executed in the scalar unit. They place scalar operands required for vector execution in the scalar operand queues that are

between the scalar and vector units. This allows scalar address calculations that are required for vector execution to complete independently of vector execution.

The vector processing unit is designed to allow vector load/store instructions to execute decoupled from vector execute unit 20. The vector load/store unit 22 issues and executes vector memory references when it has received the instruction and memory operands from scalar processing unit 12. Vector load/store unit 22 executes independently from vector execute unit 20 and uses load buffers in vector execute unit 20 as a staging area for memory load data. Vector execute unit 20 issues vector memory and vector operations from instructions that it receives from scalar processing unit 12.

When vector execution unit 20 issues a memory load instruction, it pulls the load data from the load buffers that were loaded by vector load/store unit 22. This allows vector execution unit 20 to operate without stalls due to having to wait for load data to return from main memory 16.

A method for reducing delays when synchronizing the memory references of multiple processors (such as processors 12 and 14) will be discussed next. The method is useful when a processor is performing writes that, due to default memory ordering rules or an explicit synchronization operation, are supposed to be ordered before subsequent references by another processor.

It is often the case that the address for a write operation is known many clocks (perhaps 100 or more) before the data for the write operation is available. In this case, if another processor's memory references must be ordered after the first processor's writes, then a conventional system may require waiting until the data is produced and the write is performed before allowing the other processor's references to proceed.

It is desirable to split the write operations up into two parts-- a write address request and a write data request--and send each out to memory system 16 separately. One embodiment of such a method is shown in Fig. 2. In the embodiment shown in Fig. 2, write address requests are sent to memory 16 at 50, where they are held in the memory system at 52, either by changing the state of the associated cache lines in a cache, or by saving them in some structure. The purpose of the write address request is

to provide ordering of the write request with subsequent requests. Once the write address request has been sent out to the memory system, requests from other processors that are required to be ordered after the write can be sent out to the memory system, even though the data for the write request has not yet been produced.

As the subsequent requests by other processors are processed by the memory system, they are checked at 54 against the stored write addresses. If, at 56, there is no match, then the subsequent requests can be serviced immediately at 60. If, however, there is a match at 56, control moves to 58, where the requests are held in the memory system until the write data arrives, and then serviced.

Not all stores have to be ordered with other memory references. In many cases, the compiler knows that there is no possible data dependence between a particular store reference and subsequent references. And in those cases, the references proceed it just lets the hardware do its own thing and the two references may get re-ordered.

Where, however, the compiler thinks that there may be a dependence, computer system 10 must make sure that a store followed by a load, or a load followed by a store, gets ordered correctly. In one embodiment, each processor 12 and 14 includes an instruction for coordinating references between processors 12 and 14. One such synchronization system is described in Patent Application No. xx/zzz,zzz, entitled "Multistream Processing System and Method", filed on even date herewith, the description of which is incorporated herein by reference.

In one embodiment, computer system 10 takes the store address and runs it past the other processor's data cache to invalidate any matching entries. This forces the other processor to go to memory 16 on any subsequent reference to that address.

Processor 12 then sends the store addresses out to memory 16 and saves the addresses in memory 12. Then, when another processor 12 (or 14) executes a load that would have hit out of the data cache, it will miss because that line has been invalidated. It goes to memory 16 and gets matched against the stored store addresses. If the reference from the other processor does not match one of the store addresses stored in memory 16, it simply reads its corresponding data from memory. If it does, however,

match one of the store addresses stored in memory 16, it waits until the data associated with that store address is written. Memory 16 then reads the data and returns it to the processor that requested it.

The method of the present invention therefore minimizes the delay waiting for the write data in the case there is an actual conflict, and avoids the delay in the case when there is not a conflict.

As an example, consider the case where processor A performs a write X, then processors A and B perform a synchronization operation that guarantees memory ordering, and then processor B performs a read Y. The method of the present invention will cause processor A to send the address for write X out to the memory system as soon as it is known, even though the data for X may not be produced for a considerable time.

Then, after synchronizing, processor B can send its read Y out to the memory system. If X and Y do not match, the memory system can return a value for Y even before the data for X has been produced. The synchronization event, therefore, did not require processor B to wait for processor A's write to complete before performing its read.

If, however, read Y did match the address of write X, then read Y would be stalled in the memory system until the data for write X arrived, at which time read Y could be serviced.

In one embodiment, even though the write data and write address are sent at different times, they are received in instruction order at memory 16. In such an embodiment, you don't have to send an identifier associating an address with its associated data. Instead, the association is implied by the ordering.

In one embodiment, memory 16 includes a store address buffer 19 for storing write addresses while the memory waits for the associated write data.

The method according to the present invention requires that the participating processors share a memory system. In one embodiment, the processors share a cache, such as is done in chip-level multiprocessors (e.g., the IBM Power 4). In one such embodiment, store address buffer 19 is located within the cache.

In the embodiment shown in Fig. 1b, vector stores execute in both the vector load/store unit 22 and the vector execute unit 20. As noted above, the store addresses are generated in the vector load/store unit 22 independently of the store data being available. The store addresses are sent to memory 16 without the vector store data. When the store data is generated in vector execute unit 20, the store data is sent to memory 22 where it is paired up with the store address.

The method for reducing delays when synchronizing the memory references of multiple processors can be extended as well to multiple units within a single processor (such as the vector and scalar units of a vector processor).

A computer 10 having a processor 28 connected across an interconnect network 18 to a memory 16 is shown in Fig. 3. Processor 28 includes three functional units, all of which share access to memory 16. Vector processing computer 10 in Fig. 3 includes a processor 28. Processor 28 includes a scalar processing unit 12 and two vector processing units (14.0 and 14.1). Scalar processing unit 12 and the two vector processing units 14 are connected to memory 16 across interconnect network 18. In the embodiment shown, memory 16 is configured as cache 24 and distributed global memory 26. Vector processing units 14 include a vector execution unit 20 connected to a vector load/store unit 22. Vector load/store unit 22 handles memory transfers between vector processing unit 14 and memory 16.

In the embodiment shown in Fig. 3, store address buffer 19 is stored in cache 24. In contrast to commercial microprocessors, which store write addresses locally in order to compare them to subsequent accesses to the same memory location, system 10 in Fig. 3 keeps store address buffer 19 in cache 24. This allows synchronization across more than one processor and/or more than one decoupled functional unit executing in a single processor.

For instance, in one embodiment, four processors 28 and four caches 24 are configured as a Multi-Streaming Processor (MSP) 30. An example of such an embodiment is shown in Fig. 4. In one such embodiment, each scalar processing unit 12 delivers a peak of 0.4 GFLOPS and 0.8 GIPS at the target frequency of 400 MHz.

Each processor 28 contains two vector pipes, running at 800 MHz, providing 3.2 GFLOPS for 64-bit operations and 6.4 GFLOPS for 32-bit operations. The MSP 30 thus provides a total of 3.2 GIPS and 12.8/25.6 GFLOPS. Each processor 28 contains a small Dcache used for scalar references only. A 2 MB Ecache 24 is shared by all the processors 28 in MSP 30 and used for both scalar and vector data. In one embodiment processor 28 and cache 24 are packaged as separate chips (termed the "P" chip and "E" chips, respectively).

In one embodiment, signaling between processor 28 and cache 24 runs at 400 Mb/s on processor-cache connection 32. Each processor to cache connection 32 shown in Fig. 4 uses an incoming 64-bit path for load data and an outgoing 64-bit path for requests and store data. Loads can achieve a maximum transfer rate of 51 GB/s from cache 24. Stores can achieve up to 41 GB/s for stride-one and 25 GB/s for non-unit stride stores.

In one embodiment, global memory 26 is distributed to each MSP 30 as local memory 48. Each Ecache 24 has four ports 34 to M chip 42 (and through M chip 42 to local memory 48 and to network 38). In one embodiment, ports 34 are 16 data bits in each direction. MSP 30 has a total of 25.6 GB/s load bandwidth and 12.8-20.5 GB/s store bandwidth (depending upon stride) to local memory.

In the embodiment shown in Fig. 4, the store address buffer could be stored in either cache 24 or shared memory 26. This allows synchronization across more than one processor 28 and/or more than one decoupled functional unit executing in a single processor 28.

In some systems, a load needed to produce store data could potentially be blocked behind a store dependent on that data. In such systems, processors 28 must make sure that loads whose values may be needed to produce store data, cannot become blocked in the memory system behind stores dependent on that data. In one embodiment of system 10, processing units within processor 28 operate decoupled from each other. It is, therefore, possible, for instance, for a scalar load and a vector store to occur out of order. In such cases, the processor must ensure that load request which

occur earlier (in program order) are sent out before store address requests that may depend upon the earlier load results. In one embodiment, therefore, issuing a write request includes ensuring that all vector and scalar loads from shared memory for that processor have been sent to shared memory prior to issuing the write request.

In one embodiment, the method according to the present invention is used for vector write operations, and provides ordering between the vector unit 14 and the scalar unit 12 of the same processor 28, as well as between the vector unit of one processor 28 and both the vector and scalar units of other processors 28.

Write addresses could be held by the memory system in several different formats. In one embodiment, a write address being tracked alters the cache state of a cache line in a shared cache within a processor 28. For example, a cache line may be changed to a "WaitForData" state. This indicates that a line contained in the cache is in a transient state in which it is waiting for write data, and is therefore inaccessible for access by other functional units.

In another embodiment, a write address being tracked alters the cache state of cache line in cache 24. For example, a cache line may be changed to a "WaitForData" state. This indicates that a line contained in cache 24 is in a transient state in which it is waiting for write data, and is therefore inaccessible for access by other processors 28.

In another embodiment, write addresses to be tracked are encoded in a structure which does not save their full address. In order to save storage space, the write addresses simply cause bits to be set in a bit vector that is indexed by a subset of the bits in the write address. Subsequent references check for conflicts in this blocked line bit vector using the same subset of address bits, and may suffer from false matches. For example, a write address from one processor to address X may cause a subsequent read from another processor to address Y to block, if X and Y shared some common bits.

In an alternate embodiment of such an approach, a write address being tracked is saved in a structure that holds the entire address for each entry. Subsequent references check which detect a conflict with an entry in the blocked line bit vector, access the

structure to obtain the whole write address. In this embodiment, only true matches will be blocked.

This invention can be used with multiple types of synchronization, including locks, barriers, or even default memory ordering rules. Any time a set of memory references on one processor is supposed to be ordered before memory references on another processor, the system can simply ensure that write address requests of the first processor are ordered with respect to the other references, rather than wait for the complete writes, and the write addresses can provide the ordering guarantees via the matching logic in the memory system.

The method according to the present invention reduces latency for multiprocessor synchronization events, by allowing processors to synchronize with other processors before waiting for their pending write requests to complete. They can synchronize with other processors as soon as their previous write request addresses have been sent to the memory system to establish ordering.

Definitions

In the above discussion, the term "computer" is defined to include any digital or analog data processing unit. Examples include any personal computer, workstation, set top box, mainframe, server, supercomputer, laptop or personal digital assistant capable of embodying the inventions described herein.

Examples of articles comprising computer readable media are floppy disks, hard drives, CD-ROM or DVD media or any other read-write or read-only memory device.

Portions of the above description have been presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the ways used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities

take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar computing device, that manipulates and transforms data represented as physical (e.g., electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is calculated to achieve the same purpose may be substituted for the specific embodiment shown. This application is intended to cover any adaptations or variations of the present invention. Therefore, it is intended that this invention be limited only by the claims and the equivalents thereof.